ADDRESS SIZE INDEPENDENCE IN A 16-BIT MINICOMPUTER

Philip E. Stanley
Honeywell Informations Systems, Inc.
300 Concord Road
Billerica, MA 01821

ABSTRACT

Most minicomputers do not distinguish in their architecture between addresses and other operands, with the result that operand size becomes a de facto limit on address size (and thus memory size). This has created serious problems for many architectures when attempting to build systems with large address space.

It is possible, however, to make address size "invisible" to the programmer and independent of operand/word size. This is achieved by defining the architecture to segregate addresses from other operands, and to compensate for the number of storage words required to hold an address.

In a system so designed, the assembly source code written by a programmer then becomes "independent" of the address size of the particular computer for which it is intended; differences in address size becoming the concern of only the assembler.

The implementation of this concept in a new minicomputer architecture is described. The necessary features in the architecture are identified, and the manner in which the assembler treats addresses is described.

The effects upon ease of programming of these features is considered, and the object code produced (from a single source code) for two systems with different address sizes is analyzed. It is found that the sample programs will be only 5% larger and 3% slower for a system with two-word addresses than for a system with one-word addresses.

I - INTRODUCTION

One of the most difficult (and most recurrent) problems facing the minicomputer architect is that of sufficient address space. Addresses should be potentially large enough so that all of the data and procedures for most processes can be directly addressed without recourse to memory mapping. Memory mapping/management is not inherently undesirable, but its primary purpose is to solve other problems than that of inadequate address size (Randall, 1968). Such large addresses, however, should not compromise the architectural simplicity and minimal hardware cost which characterize the minicomputer.

A minicomputer is usually organized around a fixed, relatively small memory word size (typically 12 to 18 bits), and uses that word size as a common controlling parameter throughout its architecture. Operands, instructions and addresses are all usually of that same size. Frequently, both for reasons of cost, and for purposes of address manipulation, addresses and operands even share the same operating registers.

The use of such an organization can be very cost-effective, but has the serious deficiency that it creates a de facto limit on memory size, by requiring that addresses be no larger than operands. If memory word/register size is 16 bits (a common size), for example, then addresses are constrained to a maximum size of 65,536 words.

This has proven to be a severe limitation in recent years, for both functional and economic reasons. Tasks and applications have tended to expand in scope, requiring additional memory; operating systems and higher-level languages, while improving system functionality, have had a major impact on memory size. Reduced memory costs have also fueled user requirements for large memories: Memory costs decrease at the rate of 26% to 41% per year; since users tend to buy constant dollar amounts of memory, it follows that the amount of address space typically required doubles every 2-3 years (Bell, 1976).

A number of minicomputer architectures have accepted this de facto limit on memory size, and then been found wanting as the demand for larger memories increased.

One example is the Honeywell 116, 516/416, 316, System 700, Level 6/06 series of minicomputers. Originally designed to support 16K (K=1024) words of memory, this family has twice been expanded; the first time by an architectural modification which increased address space to 32K words, the second by the addition of a somewhat cumbersome bank-switching scheme which increased physical memory size to 64K words.

A second example, of more recent origin, is the PDP-11 family of computers. Initial members of this family supported an address space of 56K bytes (entirely adequate in 1969, the year of their introduction); within two years it was found necessary to introduce memory management in order to meet the demand for larger physical memories (Bell, 1976).

II - ADDRESS SIZE INDEPENDENCE

When the architectural and design studies for a new minicomputer were begun at Honeywell Information Systems 4 years ago, it was clear from the beginning that this was the development of not just one, but an entire family of compucers. This meant that the new architecture had to support not only the system with a large address base and elaborate operating systems, but also the small OEM-oriented system where speed and minimum program size were essential. Equally essential, however, was the necessity not to impair program mobility (the ability to transport programs from from machines at one end of the family to machines at the other).

A particular goal of the study was to develop a method of dealing with addressing which would not be tied to word size, but be open-ended and consistent in all members of the family, and not impair program mobility from one member of the family to another.

The word size (the number of bits of data represented by the least significant bit of an address) had already been fixed at 16 bits. If this word size represented the largest address size, however, memory size would be limited to 64K words; sufficient for present applications, but obviously inadequate for future needs. It was felt that to have a reasonable certainty of being able to satisfy memory requirements for the life of the family, that maximum memory size should be at least 8 to 10 million words (implying an address size of at least 23 bits). However, addresses larger than 16 bits would then require 2 words of memory to hold them, and take twice as long to load and store. The large-system user, running under an operating system, would not object to this burden,

ZIE VOOR TITEL BOEK; de 2e PAGINA

since it would significantly reduce the amount of storage management and overlay activity which, of necessity, accompanies a small address space. The small-system, OEM user would be unwilling to incur this space/time penalty, however, since his address space requirements are smaller, and his application is typically more cost/performance sensitive.

Examination of the nature and usage of addresses reveals some interesting properties, particularly in the ways in which they differ from other operands. All other operands are programmer visible, explicitly typed and sized; they are bits, bytes, words or multi-words, etc. The programmer is aware of the size of the item being dealt with and uses this characteristic in manipulating it. Addresses, however, are primarily the concern of the assembler, and only of secondary interest to the programmer. They are the names of structures, arrays and program locations. The programmer wants to be able to assign names as required, and then reference them as necessary, without getting immersed in address computation and manipulation. (It was the burden of such tasks that led to the development of assemblers in the first place.) To the programmer, address size is unimportant; as long as an address is "big enough" the programmer doesn't really care.

Consideration of the differences between addresses and other operands led, in turn, to a concept which we refer to as address size independence; a philosophy of architecture in which the size of addresses is essentially invisible to the programmer at the level of assembler source code, and is only apparent at the object code level.

Under this concept, the same source code can be assembled and run on any member of the family; however, the assembler used to create object code for those members whose addresses are larger than 16 bits would be different from the assembler used to create object code for the small members of the family (see Fig 1). The large system assembler would create two words of storage for each memory address in the code, in the format shown; the small system assembler only one.

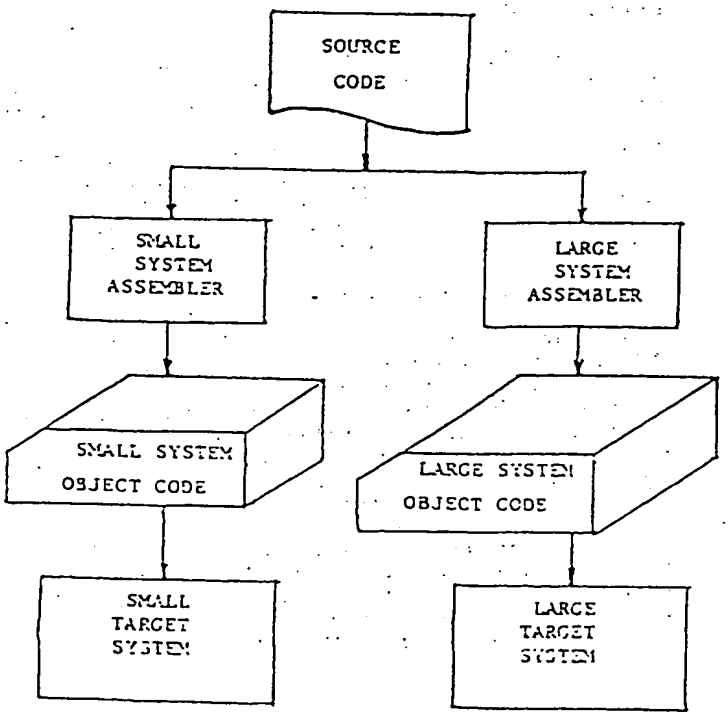

Fig 1 - Assembly Paths

The difference between the two types of systems would be the manner in which they process and use addresses; the large system would operate on the assumption that all addresses are two words in length, the small that they were one word (see Fig 2).

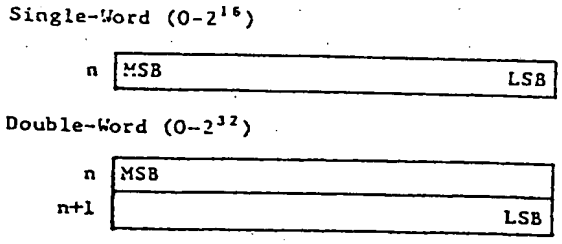

Fig 2 - Address Formats

Each system would receive object code containing addresses suitable to its hardware, while the programmer, operating at the level of the single source code from which both object codes are derived, would deal with addresses as tags and identities, in a non-size-specific fashion.

This philosophy involves a number of interacting concepts.

The first, control of address manipulation; a careful limitation of the direct accessibility to the programmer of addresses as operands. Such natural operations as loading, storing and certain types of address manipulation would be possible, while at the same time any usage or manipulation which would be cognizant of and sensitive to their size, would not be.

The second, compensation for size at time of use defining address syllable functionality in such a way that any use of or reference to an address by the programmer which is sensitive to address size will be compensated for by the hardware/firmware. (This permits such common operations as indirect addressing, stepping through a string of addresses, or indexing in an array of addresses.)

The third, awareness of misuse; the detection, wherever possible, of address manipulations by the programmer which, although technically permissible, in fact represent variant usage of the hardware, and make the program sensitive to address size.

A fourth, assembler sensitivity to address size; provision of a method whereby the assembler can be informed of the existence of addresses or address-sized items in the specification of constants and offsets and the creation of data structures, so that these items can be correctly processed into 1 or 2 words according to the size of the target system.

III - IMPLEMENTATION (HARDWARE)

In order to achieve this desired automatic handling of, and thus independence of address size from the programmer's point of view, a number of special features were required in the hardware and the architecture.

First, and perhaps most significantly, the registers in the architecture are very carefully classified and separated on the basis of whether they contain addresses, or other, non-address, operands (see Fig. 3). Those containing addresses are restricted to the sever

base registers, and the program counter. All other registers are classified as containing other operands (status and mode information, test result indicators, bits, bytes, words and multiwords, etc.), all of whose sizes and formats are very visible to the programmer. The size definition of the 8 address-containing registers is left open in the architecture description, specifying only that they are between 16 and 32 bits in length, contain integer quantities with the range 0 to 2^n-1 (where n is the length), and will be of different sizes in different configurations/family members.

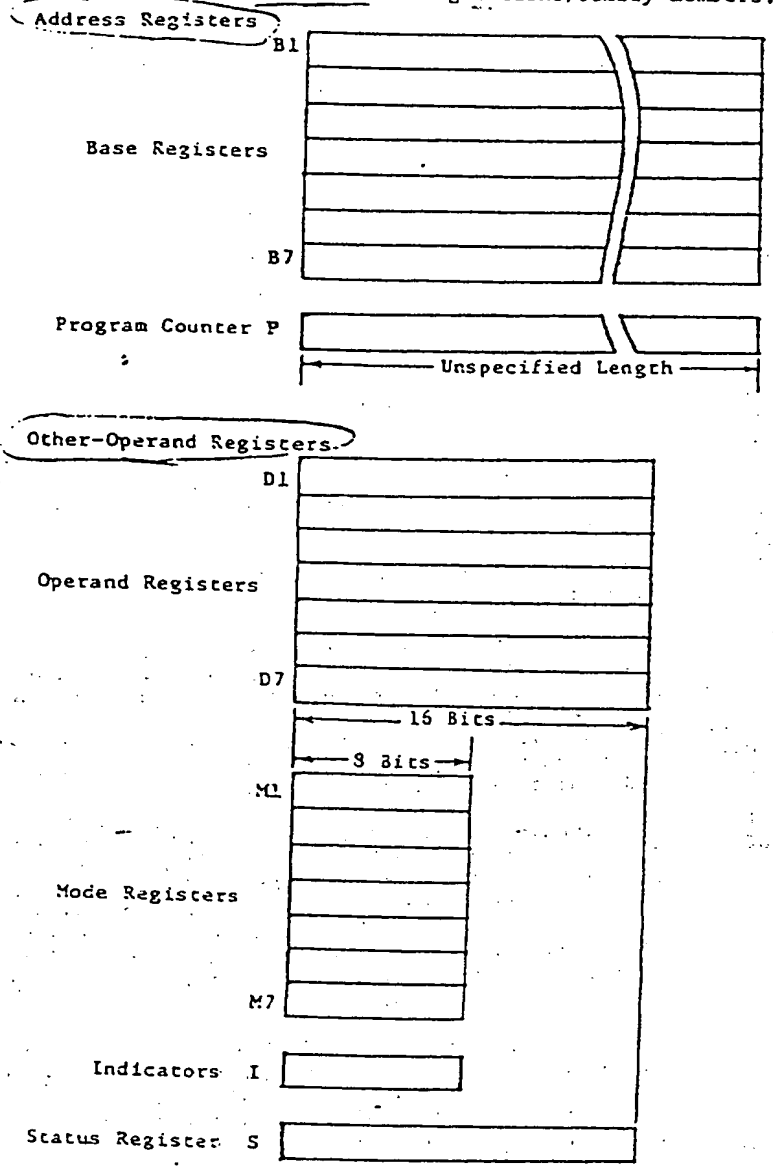

Fig 3 - System Register Set

Separating the address registers from the other operand registers implies that they have separate op codes as well, and this is used to limit the types of operations and manipulations available to the programmer to those which are address-size insensitive. Five op codes are provided for base register manipulation: Load, Store, Swap, Compare and Load Address (a special function which gives the programmer the ability to modify base registers under control of the address syllable of the instruction). An instruction is provided for testing an address for null (=0), in either a base register or memory. Two op codes are available for program counter manipulation, both Jumps. An eighth op code, Link Jump, affects both the program counter and base register.

No other op codes affect the address registers. The shift, logic, arithmetic and bit-operations available for the other-operand registers are specifically not provided; they are all operand size sensitive, and would require programmer awareness of address size.

It might be pointed out here that this separation of registers has yielded the fringe benefit of more registers and more opcodes than would have otherwise been expected in a 16-bit architecture. There is no space wasted in the op code/register type matrix for unnecessary and undesirable operations like multiplies divides, shifts, etc. on address registers; the bit combinations thus saved are used for more op codes specific to the other-operand registers, enriching the instruction repertoire.

It is not sufficient, however, to control just the explicit manipulation of addresses by the programmer; it is also necessary to treat the problem of addresses encountered during address syllable resolution.

Three types of address syllable resolution evidenced special problems when dealing with addresses in memory: global addressing, in which the address is part of the instruction; indexing, when the array element being referenced is itself an address; and moving base register addressing (base, post-incremented and base, pre-decremented) when the operand being stepped past is an address. All are sensitive to address size (1 word for $n \leq 16$, 2 words for $16 < n \leq 32$) and all are resolved by extending techniques already present in the architecture to include sensitivity to address size.

In global addressing, the address of the operand is part of the instruction; thus, in a SAF (Short Address Form) system, the instruction occupies two words, while in a LAF (Long Address Form) system, the instruction occupies three words. The architectural definition of program counter functionality was elastic enough to accommodate this difference. It had been defined as capable of automatically incrementing by an amount from one to six words to permit the use of multiword instructions; it was simple and consistent to define it as also sensitive to address size, incrementing by an extra word after the fetch of an instruction using global addressing in a LAF system.

It should be noted that the existence of new, variable length instructions does not create untoward problems for either assembler or programmer. All control transfer instructions (Jumps and Branches) are formatted to explicitly identify their (alternative) destination. No implicit assumptions, such as the skips performed by branches in some systems, exist as to destination to interfere with assembler generation of SAF or LAF code as appropriate.

When indexing into an array of addresses, address size differences are compensated for by the already-present architectural feature of "indexing to the kernel". This is a technique whereby all indexed memory references are operand size sensitive, automatically aligning the index prior to the index addition so that the least significant bit of index value represents one "item" in the array (not necessarily one word). Here again, only a minor extension was necessary to encompass SAF/LAF differences. In a SAF system, no index realignment is necessary; the least significant bit of the index represents one word. In a LAF system, the LSB of the index represents two words, and the hardware causes it to be left shifted 1 bit (doubled) prior to the index addition. This permits the compatible use of address pointers in both SAF and LAF configurations.

The third type of addressing difficulty, that involving moving (pre-decremented or post-incremented) base registers, was resolved as in the other two, by extending already existing functionaltiy so that it becomes sensitive to address size. The amount by which the base registers increment or decrement in this type of addressing was already variable, being whatever integral number of words was appropriate to the size of the operand (a function of the op code). It was, again, very simple and consistent to extend this so that when the operand was an address, the size of the increment/decrement was then sensitive to the SAF/LAF configuration.

Another class of problem relating to address size independence was that of address overflow/underflow. Totally unrelated to address size/word size disparities, overflow/underflow is a condition which occurs when address computation creates an address "below location zero", or "above" the largest address which will fit in an address register in a particular configuraion. An example of underflow would be to index by a value of -200 relative to a base address of 100; the resulting address (location "-100") cannot be represented in an address register (addresses have the range of $0 \leq EA \leq 2^n-1$), the borrow would be lost, and the result appear to be a very large address (100 words below the largest address possible for the configuration). Similarly, overflow would be to index by +200 relative to a base address of 65,436 in a system with 16-bit addresses; the resulting address (location "65,636") is too large to fit in a 16-bit address register, the carry would be lost and the resulting address appear to be 100. (Note, however, that in a different system with larger address registers this computation would have been permissible and the resulting address legitimate.)

Address overflow/underflow is clearly an area in which address size is visible to the programmer. Furthermore, there is no way to prevent the programmer from specifying address parameters which can cause it (there would, in fact, be a tendency for too-clever programmers to deliberately use this behavior to increase index/displacement range when near the bottom or top of the address space).

It was decided, therefore, to make address overflow/underflow illegal, testing for the appropriate combination of sign and carry at the time of addition, and forcing a trap (an internal synchronous interrupt) if detected. This trap invokes the same software activated by a reference to uninstalled memory (a "missing resource" trap), and allows the programmer to correct this (hopefully) inadvertent misuse.

IV - IMPLEMENTATION (SOFTWARE)

In the process of defining the handling of addresses by the hardware, it became clear that some compensation for address size would have to be made in the system software as well. Certain classes of operations could neither be compensated for nor prevented from occurring.

A case in point was the explicit definition by the programmer of either an offset or constant specifying some number of words and addresses, or the specification of a common block or data structure containing addresses. In such cases, there is no way to preclude programmer cognizance of address existence. The programmer must inform the assembler of the desired offset size; the common block must be large enough to hold the necessary data.

Here again, however, we could distinguish between address existence and address size. A special internal value label was defined for the assembler, called $AF. The value of $AF is set by the assembler, in accordance with the configuration of the target system. If for a SAF (1 word address) system, then $AF=1; if for a LAF (2 word address) system, then $AF=2.

This label, $AF, can be used by the programmer wherever necessary when referring to addresses, in order to specify number of words per address and thus distinguish SAF from LAF assembly as appropriate.

Two examples will suffice to illustrate its utility: If it is desired to displace relative to a base register by 23 items, where 18 of the items are words, and five are addresses, then the assembly language addressing statement would be:

```
....,$Bx.18+5*$AF
```

which would be assembled with a displacement of 23 for a SAF system, and 28 for a LAF system. This would be the general method for accessing into any non-homogeneous data array, such as a stack or parameter table.

Similarly, the reserve statement

```
(LABEL)     RESV    12*$AF,0
```

intended to reserve space for 12 addresses at location (LABEL), will reserve 12 words in a SAF system, but 24 in a LAF. This technique allows the programmer to allocate space for address arrays in SAF/LAF independent fashion.

V - RESULTS & EVALUATION

The addition of the $AF label in the assembler was the last major element necessary to make address size independence a workable concept.

Other, minor, changes were still required (such as specifying hardware dedicated memory locations in a manner which made them readily accessible in both SAF and LAF) but these were not critical. The separate registers and op codes, the automatic compensation for address size, the address overflow/underflow detection, and the $AF label were the basic elements supporting address size independence.

Two questions had to be answered, however. Had ease of programming been compromised in order to introduce this functionality? and how successful were these features in supporting SAF/LAF independence and program mobility?

Ease of programming is a difficult property to evaluate and quantify, being to a large extent subjective and programmer-dependant. In this case, where we were concerned primarily with second-order differences, it was doubly so. What was at issue was: how restrictive, how hard to learn and how artificial in appearance, were the special features in the architecture which had been developed to support SAF/LAF independence. It was this "ease of programming" which had to be evaluated.

A significant insight into this aspect of the architecture was achieved by composing the list of guidelines for programmers writing SAF/LAF independent code. It is obvious that the more complex and numerous such guidelines are, the less easy and natural to program is the architecture. In this case, the list of guidelines was short, simple and not markedly restrictive, and could even be considered as guidelines for good programming practice. They are largely concerned with awareness of the existence of addresses as compared to

other operands, and have been found to be neither difficult to learn nor to follow (a summary of these guidelines will be found in Appendix A).

A second test of ease of programming was to examine the coding of the various benchmark programs, code kernels which exercise and test individual computer features, searching for strengths and deficiencies. Such benchmarks are common in architecture evaluation (Fuller, 1977), and a set of 10-15 such has been used for this purpose at Honeywell for a number of years. Different benchmarks test different architectural features, from byte handling to argument passing, but it was not specific, overt features that were of interest in this case, so much as an illustration of the programming difficulties and anomalies which they exhibited due to the SAF/LAF independence.

Here again, it seemed we had been fairly successful in avoiding awkward functional situations. It was hard to identify any specific case where the special SAF/LAF functionality (or progra-ming guidelines) cost us either space or speed. As an example, consider the benchmark program, HIST (Histogram), designed to test a computer's looping and comparison facilities. This subroutine tests an array of 1000 integers, ITABLE, whose address is procedural in the calling routine, and creates a histogram of their integer values (discarding all values less than zero and greater than 100) in an integer array, OTABLE, whose address is procedural in the subroutine and is returned to the calling routine in base register B2. The source code for this subroutinetine, together with its SAF Assembly, is shown in Appendix B.

Note that there are only three occurrences of addresses in this coding, two in the call (<HIST and <ITABLE) and one in the subroutine (<OTABLE). Note also the use of the Link (B7) to get <ITABLE and step past it at the same time (at location 1007), and the automatic stepping past of the Program Counter when using the occurrence of <OTABLE (at location 1000). There are no identifiable places in this subroutine where the features supporting address size independence penalize, or even become noticeable to the programmer.

On the basis of: a) Consideration of the guidelines for writing SAF/LAF independent code, and b) Examination of the coded bench mark programs used for evaluating the architecture, we were able to conclude that SAF/LAF independence did not significantly degrade ease of understanding or programming in the architecture. This has subsequently been substantiated by the experience of the large body of programmers now using this machine; the general consensus being that the one barrier to ease of programming in this system is its exceptionally rich instruction set and address syllable, not the concept of SAF/LAF independence.

The second question mentioned earlier, SAF/LAF independence and Program Mobility can probably best be answered by considering the LAF Assembly of the benchmark program, HIST, cited earlier (Appendix C).

As expected, the differences between the SAF and LAF assemblies is minor, and of the type automatically compensated for by the hardware. The number of words required for the subroutine has increased from 17 to 18 words (a 6% increase), and the number of memory cycles (for an ITABLE with 100 integers out of range) from 8912 to 8915 (less than 0.1%).

What is more important than size and speed, however, is to note how easy it is to write programs that can assemble in either mode; no special precautions were taken in coding this benchmark to provide for LAF addressing; the use of the existing functionality did did that automatically and without problems.

Prior to coding the benchmark programs, it had been estimated that the object code for a LAF system would be 5 to 10 percent larger and run 5 to 10 percent slower than that for a SAF system. This estimate has proven to be conservative; the coded benchmarks indicate that typical size and speed differences are 5 percent and 3 percent respectively (see Table 1).

BENCHMARK FUNCTION	% INCREASE SAF:LAF EXECUTION TIME	MEMORY UTILIZATION
ASCII to Hollerith Conversion	9.6	5.8
BCD to Binary Conversion	1.1	4.0
Unpack and Edit	.4	.2
Communications Buffer Driver	7.0	4.7
Histogram	0.1	6.3
Memory to Memory Move	1.0	2.6
Serial Byte Key Search	2.4	5.6
Tolerance Check	2.5	.8
Quicksort	1.4	12.0
Average	2.8	4.7

Table 1 - LAF Size/Speed Increase

The differences between SAF and LAF are equally small in large segments of operational code. Consider the following two examples (both program modules from the Level 6 GCOS/BES software):

The system executive in GCOS/BES is a good example of a large (6000 word) program module with many external references and absolute addresses, yet the size difference between SAF and LAF object programs for this module is only 5.4%

The GCOS/BES Text Editor is an example of a large (4000 word) program with almost no external references or absolute addresses. For this program, the LAF object code is only 20 words larger than the SAF (0.5%).

Less exact information is available as to execution time differences between the two modes for system software. Measurements of software systems performance are inconclusive, since most operational software tends to be frequently I/O bound, masking the differences in program execution time caused by SAF/LAF distinctions. What data is available, however, indicates that the 3% speed penalty observed for the benchmarks is equally valid for operational software as well.

ACKNOWLEDGEMENT

The concept of address size independence is the invention of no one individual, but rather the synthesis of ideas and suggestions from many. The contributions of John Grandmaison, Richard Lemay, Patrick Prange and William Woods are particularly acknowledged.

REFERENCES

1. (no cited author), "32-Bit Minis: Tempest In a Teapot?", Modern Data 9,3 (March, 1976), pp 41-43

2. (no cited author), "Assembly Language Program Independence", GCOS 6 Program Preparation Manual (CB06), Honeywell Information Systems, Billerica, MA, 1978

3. Amdahl, G. M.; Blaauw, G. A.; and Brooks, F. P. Jr., "Architecture of the IBM System/360", IBM Journal of Research and Development, 8,2 (1964), pp 87-101

4. Bell, Gordon; and Strecker, William D., "Computer Structures: What Have We Learned From The PDP-11?", ACM/IEEE Symposium On Computer Architecture, 1976, pp 1-14

5. Denning, Peter J., "Third Generation Computing Systems", Computing Surveys 3,4 (December, 1971), pp 175-216

6. Fuller, Samuel H., "Price/Performance Comparison of C.mmp and the PDP-10", ACM/IEEE Symposium On Computer Architecture, 1976, pp 195-202

7. Fuller, Samuel H.; Stone, Harold S.; Burr, William E., "Initial Selection and Screening of the CFA Candidate Computer Architectures", AFIPS Conference Proceedings, Vol. 46, 1977 National Computer Conference, pp 139-146

8. Fuller, Samuel H.; Shaman, Paul; Lamb, David; Burr, William E., "Evaluation of Computer Architectures Via Test Programs", AFIPS Conference Proceedings, Vol. 46, 1977 National Computer Conference, pp 147-160

9. Randall, B.; and Keuhner, C. J., "Dynamic Storage Allocation Systems", Communications of the ACM 11,5 (May, 1968), pp 297,306

10. Tennenbaum, Andrew S., "Ambiguous Machine Architecture and Program Efficiency", Computer Architecture News (SIGARCH), 6,3 (August, 1977), pp 11-13

11. Wegner, P., "Programming Languages, Information Structures, and Machine Organization", McGraw-Hill, New York, 1968

12. Yuen, C. K., "The Rise and Fall of General-Purpose Registers", Computer, 10, 10 (October, 1977), pp 85-86